Spring 2005 - Programming Languages Qualifier

Computer Science Department
University of Massachusetts Lowell
Lowell, MA 01854

Feb. 6, 2005.

There are 11 problems on this exam with a total value of 120 points. You must work Problem 1. Select 8 out of the next 10 problems (1 to 11). If you attempt more than 8 out of the other 10 problems then clearly indicate in your blue books the 8 problems that should be graded.

You should write the answers to problems 1–5 in one blue book and the answers to problems 6–11 in the other blue book.

1. **Programming language concept distinctions (20 pts).** Briefly describe the similarities and/or differences between the following concepts:

   (a) self (a.k.a. this) and super.
   (b) continuations and exceptions.
   (c) abstract data types and objects.
   (d) abstract data types and modules.
   (e) lexical (also known as static) scope and dynamic scope.
   (f) Horn clause and unit clause.
   (g) Operational semantics and denotational semantics.

2. **Scheme (10 pts).**

   Given the definition

   \[
   \text{(define (cons x y)}
   \text{ (lambda (m))}
   \text{ (cond ((eq? m 'my-car) x)}
   \text{ ((eq? m 'my-cdr) y)}
   \text{ (else (error "illegal list operation attempted")))))
   \]

   (a) (4 pts) define car and cdr to work with this definition of cons
   (b) (6 pts) extend the definition of cons above as needed and define set-car! and set-cdr!
       to work with your extended definition.
3. Fixed points (10 pts).

Consider the function defined recursively as

\[ f = \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 2 \times f(f(n - 1)) \]

The function \( f \) can also be defined as a fixed point of the following function

\[ F = \lambda g. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } 2 \times g(g(n - 1)) \]

(a) (4 pts) Show that the following mathematical partial function satisfies the recursion equation for \( f \) (and is thus a fixed point of \( F \)):

\[ h = \{(0, 1), (1, 0), (2, 2), (3, 4)\} \]

(b) (6 pts) Show how to find the least fixed point of \( F \) using Tarski’s method of making a sequence of approximations. Make sure that you write down the function that you find using Tarski’s method.

4. Lambda calculus (10 pts).

(a) (2 pts) What property must a closed lambda term satisfy to be a fixpoint combinator?

(b) One common fixpoint combinator in \( \lambda \)-calculus is “Curry’s paradoxical combinator”

\[ Y \equiv (\lambda f. (\lambda x. f(xx))((\lambda x. f(xx))) \]

i. (4 pts) What result do you get, using the \( F \) in the previous problem, if you calculate \((\text{Y}F)0\) using normal-order (leftmost-outermost) \( \beta \)-reduction and the usual rules for if then else and arithmetic?

Show the first 4 \( \beta \)-reduction steps in this calculation; use the abbreviation \( F \) wherever possible.

ii. (4 pts) What result do you get if you calculate \((\text{Y}F)0\) using applicative-order (leftmost-innermost) \( \beta \)-reduction and the usual rules for if then else and arithmetic?

Show the first 4 \( \beta \)-reduction steps in this calculation; use the abbreviation \( F \) wherever possible.

5. Polymorphism (10 pts).

Describe each of the following forms of polymorphism, name a language that implements it, show a short example of that sort of polymorphism in the language you have named.

(a) (3 pts) parameteric polymorphism

(b) (4 pts) subtype polymorphism

(c) (3 pts) ad-hoc polymorphism
6. ML programming (algorithm from logic programming) (10 pts).

The negation normal form \( \text{nff}(p) \) of a proposition \( p \) is logically equivalent to \( p \), and is obtained by repeatedly driving negations inward until negations are applied only to atoms. Consider the definition of the negation normal form function \( \text{nff} \). Fill in the following blanks:

\[
\text{datatype prop } = \\
\quad \text{Atom of string} \\
\quad | \text{Neg of } \quad \quad (*1*) \\
\quad | \text{Conj of prop \* prop} \\
\quad | \text{Disj of prop \* prop};
\]

\[
\text{fun nff (Atom a) } = \\
\quad | \text{nff (Neg (Atom a)) } = \text{Neg (Atom a)} \\
\quad | \text{nff (Neg (Neg p)) } = \\
\quad | \text{nff (Neg (Conj(p,q))) } = \\
\quad | \text{nff (Neg (Disj(p,q))) } = \text{nff (Conj(Neg p, Neg q))} \\
\quad | \text{nff (Conj(p,q)) } = \text{Conj(nff p, nff q)} \\
\quad | \text{nff (Disj(p,q)) } = \\
\text{val nff = fn : prop \to prop}
\]

\[
\text{val round = Atom "round"} \\
\quad \text{and square = Atom "square"};
\]

\[
\text{val round = Atom "round" : prop} \\
\text{val square = Atom "square" : prop}
\]

\[
\text{nff round;} \\
\text{val it = Atom "round" : prop} \\
\quad \text{nff(Neg(round))};
\]

\[
\text{val it = Neg (Atom "round") : prop} \\
\quad \text{nff(Neg(Neg(square)))};
\]

\[
\text{val it = Atom "square" : prop}
\]
7. Parameter Passing (10 pts).
   Consider the following C program:

   ```c
   int F (int x, int y) {
     x = x + 2;
     y = y * 2;
     return (x - y);
   }

   ...

   {
     int a = 2;
     int b = F(a, a);
     printf ("a = %d, b = %d\n", a, b);
   }

   ...
   
   (a) (2 pts) Describe the pass-by-value calling convention used by C, and show what the
   program fragment above would print.
   (b) (4 pts) Describe the pass-by-reference calling convention. Show what the program frag-
   ment would print if C implemented pass-by-reference.
   (c) (4 pts) Describe the pass-by-value/return calling convention. Show what the program
   fragment would print if C implemented pass-by-value/return.

8. Environments of Evaluation (10 pts). Consider the following Scheme code in which one
function is passed as an argument to another function. Note that the function called `pass-me`
is variadic.

   `(define pass-me
      (lambda args
        (apply + args)))`

   `(define calculate
      (lambda (proc a b c)
        (proc a b c)))`

   Using the diagramming notation of either Manis & Little or Abelson & Sussman, show the
complete environment of evaluation during evaluation of `(calculate pass-me 3 5 7)`
9. Programming (10 pts). You have the following 5 languages to choose from: C, C++, Scheme, ML, Prolog, and Smalltalk. In the language of your choice, write a **complete** program that encodes the Deterministic Finite Automaton given in the figure below:

![DFA Diagram](image)

The alphabet is $\Sigma = \{0, 1\}$; the set of states is $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$; the transition function $\delta : Q \times \Sigma \rightarrow Q$ can be read off the diagram, as well as the set of accepting states $F = \{q_3\}$. The function (or relation) performing the work - and which takes as parameters a list of binary digits corresponding to the string to be processed, and the start state $q_0$ - **must** be reusable over any such DFA, requiring only a separate re-encoding of the state transition function and the set of accepting states.

Hint on language choice: the co-author of this exam who did not propose this question solved it in 15 lines of Prolog, and thinks that it would take considerably more code to use a different language. If you are a very fluent programmer in one of the other languages you might manage in the time allotted.

10. Unification (10 pts). Explain and contrast how unification is used in ML and in Prolog. In particular, use the ML function

```ml
fun addThem [] = 0
  | addThem (x::y) = x + (addThem y)
```

and the Prolog rule

```
addThem([], 0).
addThem([X|Y], Z) :- addThem(Y, W), Z is X + W.
```

to show how unification is used and to illustrate the differences.
11. **Operational Semantics (10 pts).** Scheme supports both an environment $\rho$ and a store $\sigma$, since it allows for mutation. Recall the syntax of \texttt{let} and \texttt{let*}:

\[
\text{(let ((x1 e1) \ldots (xn en)) <body>)}
\]
\[
\text{(let* ((x1 e1) \ldots (xn en)) <body>)}
\]

The first binds the value of each $e_i$ to the corresponding $x_i$ without any knowledge of the remaining $x_j, j \neq i$, introduced in the \texttt{let}, while the second allows for any expression $e_i$ to refer to any $x_j$ introduced in the \texttt{let*} as long as $j < i$. (70 \%) Using semantic rule notation, describe the operational semantics of the two expressions, \texttt{let} and \texttt{let*}. Make sure you indicate the changes of both environment $\rho$ and store $\sigma$. To remind you of the notation, the conclusion of both semantic judgments looks like

\[
(\text{LET?}((x_1, e_1, \ldots, x_n, e_n), \rho, \sigma) \Downarrow (v, \sigma'))
\]

where the ? could be nothing or *.

(30 \%) Give an example where the single change from \texttt{let} to \texttt{let*} produces a different - non-error - result for the execution of <body>. 

6